Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities

Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, Denys Poshyvanyk

Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities

Martin White*, Michele Tufano*, Matías Martínez†, Martin Monperrus‡, and Denys Poshyvanyk*
*College of William and Mary, Williamsburg, Virginia, USA
Email: {mgwhite, mtufano, denys}@cs.wm.edu
†Université Polytechnique Hauts-de-France, Valenciennes, France
Email: matias.martinez@uphf.fr
‡KTH Royal Institute of Technology, Stockholm, Sweden
Email: martin.monperrus@csc.kth.se

***Abstract*—In the field of automated program repair, the redundancy assumption claims large programs contain the seeds of their own repair. However, most redundancy-based program repair techniques do not reason about the repair ingredients—the code that is reused to craft a patch. We aim to reason about the repair ingredients by using code similarities to prioritize *and transform* statements in a codebase for patch generation. Our approach, DeepRepair, relies on deep learning to reason about code similarities. Code fragments at well-defined levels of granularity in a codebase can be sorted according to their similarity to suspicious elements (i.e., code elements that contain suspicious statements) and *statements can be transformed* by mapping out-of-scope identifiers to similar identifiers in scope. We examined these new search strategies for patch generation with respect to effectiveness from the viewpoint of a software maintainer. Our comparative experiments were executed on six open-source Java projects including 374 buggy program revisions and consisted of 19,949 trials spanning 2,616 days of computation time. DeepRepair's search strategy using code similarities generally found compilable ingredients faster than the baseline, jGenProg, but this improvement neither yielded test-adequate patches in fewer attempts (on average) nor found significantly more patches (on average) than the baseline. Although the patch counts were not *statistically* different, there were notable differences between the nature of DeepRepair patches and jGenProg patches. The results show that our learning-based approach finds patches that *cannot* be found by existing redundancy-based repair techniques.**

***Index Terms*—software testing and debugging, program repair, deep learning, neural networks, code clones, language models**

I. Introduction

In the field of automated program repair, many repair approaches are based on a common intuition: a patch can be composed of source code residing elsewhere in the repository or even in other projects. These approaches are called redundancy-based repair techniques, since they leverage redundancy and repetition in source code [1]–[7]. For example, GenProg [8], [9] reuses existing code from the same codebase, whereas CodePhage [10] transplants checks from one application to another. This intuition has been examined by at least two independent empirical studies, showing that a significant proportion of commits are indeed composed of existing code [6], [7].

The code that is reused to craft a patch is called a *repair ingredient*. Most redundancy-based repair techniques harvest repair ingredients at random and do not reason further about the optimal strategies to select repair ingredients. In other words, these repair techniques simply brute-force search for viable ingredients, randomly searching the codebase in a straightforward trial-and-error process. Although these techniques are able to find patches, their naive search and rigid application of repair ingredients means *patches that use novel expressions are unattainable*.

In this paper, we aim to improve the reasoning about repair ingredients before they enter the repair pipeline (i.e., ingredient insertion, compilation, and test execution). Our key intuition is that a good ingredient does not come from just *any* location in the program but comes from *similar* code. We design our experiments to evaluate whether an approach built on this intuition can effectively improve patch generation.

We rely on deep learning to reason about code similarities. Deep learning has provided the software engineering research community with new ways to mine and analyze data to support tasks [11]–[20]. Our learning-based approach automatically creates a representation of source code that accounts for the structure *and meaning* of lexical elements. Our approach is completely unsupervised, and no top-down specification of features is made beforehand. We use the learned features to compute distances between code elements to measure similarities, and we use the similarities to intelligently select *and adapt* repair ingredients to generate a series of statement-level edits for program repair.

We design, implement, and evaluate DeepRepair, an approach for sorting and transforming program repair ingredients via deep learning code similarities. Our approach is implemented on top of Astor [21], a Java implementation of GenProg. DeepRepair uses recursive deep learning [22] to prioritize repair ingredients in a fix space and to select code that is most similar when choosing a repair ingredient for patch generation. *Additionally, DeepRepair addresses a major limitation of current redundancy-based repair techniques that do not adapt repair ingredients by automatically transforming ingredients based on lexical elements' similarities*. For instance, given the simple repair ingredient `return FastMath.abs(y-x) <= eps;` the transformation may involve replacing the variable eps (which is out of scope at the modification point) with the variable SAFE_MIN (in scope at the modification point).

We assess the effectiveness of using code similarities to sort or transform repair ingredients for patch generation. To do so, we evaluate our approach along several dimensions to measure what aspects contribute and do not contribute to improved repair effectiveness. We compute a baseline; evaluate sorting (at different levels of granularity and scope) in isolation; evaluate transforming in isolation (at different levels of granularity and scope); and evaluate sorting with the ability to transform repair ingredients (at different levels of granularity and scope). We evaluate everything on six open-source Java projects including 374 buggy program revisions in Defects4J database version 1.1.0 [23], [24]. In summary, our evaluation consists of 19,949 trials spanning 2,616 days of computation time. DeepRepair's search strategy using code similarities generally found compilable ingredients (Sec. III-C) faster than the baseline, jGenProg [25], but this improvement neither yielded test-adequate patches in fewer attempts (on average) nor found significantly more patches (on average) than the baseline. Although the patch counts were not *statistically* different, there were notable differences between the nature of DeepRepair's patches and jGenProg's patches. To sum up, we make the following noteworthy contributions:

- a novel learning-based algorithm to intelligently select and adapt repair ingredients for redundancy-based repair;
- the implementation of our approach for Java in a publicly available tool;
- an evaluation protocol for redundancy-based repair, including a set of novel metrics that are specific to the analysis of ingredient selection strategies;
- an evaluation on 374 real bugs from the Defects4J benchmark showing that our algorithm finds patches that cannot be found by existing redundancy-based techniques;
- online appendix with code and experimental results [26].

The paper is organized as follows. Sec. II reviews background on program repair techniques and related work on the redundancy assumption. Our approach is based on two empirically validated results: redundancy is localized in the same file and many repairs do not use new tokens. Sec. III presents DeepRepair, the first approach that can sort the fix space at well-defined levels of granularity (e.g., methods and classes) and transform program repair ingredients. Sec. IV specifies our experimental design. Sec. V reports our results. Like typical repair studies, we report the number of bugs unlocked by our approach. In our study, unlocking bugs is attributed to DeepRepair's ability to expand the fix space by transforming ingredients. On the other hand, to the best of our knowledge, unlike any repair study conducted to date, we also report the statistical significance of the difference in patch counts/attempts between our approach and the baseline. In summary, DeepRepair not only patches bugs that cannot be patched by existing generate-and-validate techniques but also finds sets of test-adequate patches that are distinctly different than the baseline's patches. The intent of our statistical analysis was to responsibly qualify the key result that DeepRepair unlocks new bugs. Sec. VI discusses threats to the validity of our work. Sec. VII concludes the paper.

II. Background and Related Work

A. Automated Program Repair

Automated repair involves the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification [27]. Behavioral repair techniques change the behavior of a program under repair by changing its source or binary code [27]. For example, GenProg [8], [9], [28], which was used in several studies [29], [30], changes the behavior of a program according to a test suite (i.e., an input-output specification) by modifying the program's source code. This *generate-and-validate* technique searches for statement-level modifications to make to an abstract syntax tree (AST).

A complementary set of repair techniques leverage program analysis and program synthesis to repair programs by constructing code with particular properties [31]–[36]. For example, SemFix [31] synthesizes (side-effect free) expressions for replacing the right-hand side of assignments or branch predicates to repair programs. Angelix [34] uses guided symbolic execution and satisfiability modulo theories solvers to synthesize patches using *angelic* values, i.e., expression values that make a given test case pass. Nopol [33] either modifies an existing conditional expression or adds a precondition to a statement or block in the code.

Both types of techniques have distinct advantages. Generate-and-validate techniques have the advantage of operating at coarse granularity with the power to mutate statements. However, these techniques generally do not mutate code below statement-level granularity, so they do not change conditional expressions nor variables. Semantics-based techniques have the advantage of operating at fine granularity on expressions and variables, enabling them to synthesize a repair even if the patch code does not exist in the codebase [31]. However, they generally do not operate at higher levels of granularity, and scalability has been a key concern.

SearchRepair [37] draws from both generate-and-validate and semantics-based techniques. SearchRepair uses symbolic reasoning to search for code and semantic reasoning to generate candidate patches. Consequently, scalability is a concern, and SearchRepair has only been shown to work on small programs. In lieu of program semantics to search for similar code, we use a learning-based approach to query textually/-functionally similar code at arbitrary levels of granularity. SearchRepair uses several software systems to repair small, student programs, whereas our approach uses the program under repair, and we aim to repair real software systems. Finally, SearchRepair depends on input-output examples to describe desired behavior, whereas we automatically learn features for distinguishing code fragments.

Recently, Yokoyama et al. [38] used code similarity to select code lines in code regions similar to the faulty code regions. Our work using code similarities differs in several important respects. They used small, fixed-sized code regions of 4, 6, and 8 lines, whereas we use code regions at arbitrary levels of granularity, making it possible to map a suspicious statement (i.e., a statement suspected to contain a bug [39])

to its method or class and query similar execution contexts or similar classes. Their similarity metric analyzed the longest common subsequence between two token sequences, whereas we use a learning-based clone detector that fuses information on structure and identifiers and is capable of finding more meaningful similarities than token-based techniques [15]. They used a collection of 24 bug-fix commits and defined a code coverage metric to evaluate their approach. We use a collection of 374 reproducible bugs and implemented our learning-based approach in an automatic software repair framework to measure effectiveness. Lastly, our approach is capable of using similarities to *transform repair ingredients*.

Our approach relies on machine learning for the fix localization problem. Prophet [40] is a learning-based approach that uses explicitly designed code features to rank candidate repairs. We use representation learning [41], [42] to automatically learn how to encode fragments to detect similarities. Other approaches train on correct (student) solutions to specific programming tasks and try to learn task-specific repair strategies [43], [44]. For example, in massively open online courses, the programs are generally small and synthetic [45]. Other approaches cannot transform statements. For example, Gupta et al. [45] use an oracle that rejects a fix if it does not preserve the identifiers and keywords present in the original statement. We use similarities to map the set of out-of-scope variables to a set of variables in scope at the modification point.

B. Redundancy Assumption

Martinez et al. [6] examined a critical assumption of GenProg that certain bugs can be fixed by copying and rearranging existing code. They validated the redundancy assumption by defining a concept of *software temporal redundancy*. A commit is temporally redundant if it is only a rearrangement of code in previous commits. They measured redundancy at two levels of granularity: line- and token-level. At line-level granularity, they found most of the temporal redundancy to be localized in the same *file*. We use learning-based code clone detection, which is capable of detecting *file*-level clones, so the search will first look in the same file and *similar files*. Moreover, their token-level granularity results imply that many repairs need never invent a new token. Ergo, the tokens exist, but repair engines need to *learn* how to use them. Again, our key insight is to look to the *learning*-based code clone detection approach, which maps tokens to continuous-valued vectors called *embeddings* that we can use to measure similarities. Then we use the similarities to consider different tokens in different contexts. Finally, we draw one more bit of insight from their empirical study. The authors note a tension between working with the line pool or the token pool [6], and they characterize this tension by the combination spaces of line- (smaller combination space) versus token-level (larger combination space) granularity. The essence of our work is to suppose there exists a manifold governed by coarse-grained snippets like lines yet can be parameterized by fine-grained snippets like tokens. DeepRepair's ingredient transformation traverses this manifold to find repairs that *cannot* be found by existing redundancy-based techniques.

Barr et al. [7] examined a history of 15,723 commits to determine the extent to which the commits can be reconstructed from existing code. The grafts they found were mostly single lines, i.e., micro-clones, and they proposed that micro-clones are useful since they are the atoms of code construction [7]. The learning-based clone detection approach uses these micro-clones to compute representations for fragments at higher levels of granularity, which we use to assess similarities and prioritize statements and values to assign to code modifications.

III. Technical Approach

Our approach is organized as a pipeline comprising three phases: recognition, learning, and repair. The language recognition phase (Sec. III-A) consumes source code of the application under repair in situ and produces training data. The machine learning phase (Sec. III-B) consumes the training data and produces encoders for encoding anything from a lexical element to a class such that similarities can be detected among the encodings. The program repair phase (Sec. III-C) uses the encoders to query and transform code fragments in the codebase for patch generation. To explain our approach, we use bug Math-63 (Revision ID: d2a5bc0) from the Defects4J database version 1.1.0 [23], [24] as a running example throughout Sec. III.

A. Language Recognition Phase

Our approach begins with a program and its set of source code files. The first stage of our language recognition phase consumes this source directory and parses its contents to create an AST or *any* model for representing the code. A well-formed, typed AST is sufficient since the visitor design pattern facilitates queries on the program under analysis such as the number of files, classes, and methods in the application sources. For example, for the Math [46] library, this stage produces a model containing 459 files, 661 classes, and 4,983 methods.

The second stage of our language recognition phase consumes the model and uses a program processor to produce corpora at different levels of granularity. In this context, a program processor is simply a utility for performing a specific action. First, we create a file-level corpus by querying the model for all the files. Our program processor creates one line per code file in the corpus by printing (from left to right) the terminal symbols [47] of the corresponding syntax tree. Simultaneously, we store keys that uniquely identify the Java source code files. Likewise, we use the program processor to build corpora at other levels of granularity, e.g., classes or methods, provided there is a way to uniquely identify the code elements. For example, Math files can be identified by their path; classes can be identified by their qualified name; and methods can be identified by concatenating the qualified name of their parent type and their signature. We build corpora at different levels of granularity because (in the next phase of our approach) we mine patterns at the level of classes, methods, and even lexical elements. These representations will be the mechanisms for querying and transforming ingredients.

The third stage of our language recognition phase normalizes the corpora by mapping some or all of the literal tokens to

```
public static boolean equals(double x, double y)
   return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
```

Fig. 1: Suspicious method in Math-63's MathUtils.java

their respective type. For example, every floating point number in the Math corpus would be replaced by a generic symbol for floating point literal values. Normalizing the corpora is the last stage of recognition and staging data for learning.

B. Machine Learning Phase

The first stage of our machine learning phase involves training a neural network language model from the file-level corpus. A (statistical) language model is a probability distribution over sentences (e.g., lines of code) in a language [48]. Recently, language modeling has been used for software engineering tasks [3], [11], [15], [49]–[62]. We require a neural network to learn representations for each term in the file-level corpus [63], and recurrent neural networks in particular can serve as effective architectures for language models of both natural language corpora and source code [11], [15], [64]–[68]. These models learn from the order of terms in a corpus, imputing *embeddings* to the terms in such a way that terms used in similar ways have embeddings that are close to each other in a feature space. We use the embeddings to initialize the second learning stage.

The second stage of our machine learning phase involves training another learner to encode arbitrary streams of embeddings. We leverage work on recursive autoencoders [69], [70] and learning-based code clone detection [15]. To demonstrate the recursive learning procedure, consider a suspicious method in Math-63's class MathUtils (Fig. 1). The method's return statement is filtered to the stream of terms in Fig. 2. Then our first machine learning stage maps the stream of terms to a stream of embeddings $\{x_0, \ldots, x_7\}$ (Fig. 2). There are seven pairs of adjacent terms in Fig. 2. Each pair of adjacent terms are encoded by agglutinating the embeddings $x = [x_\ell; x_r] \in \mathbb{R}^{2n}$ multiplying x by a matrix $\varepsilon = [\varepsilon_\ell, \varepsilon_r] \in \mathbb{R}^{n \times 2n}$ adding a bias vector $\beta_z \in \mathbb{R}^n$ and passing the result to a nonlinear vector function f:

$$z = f\left(\varepsilon x + \beta_z\right) \tag{1}$$

For example, in Fig. 2, x_ℓ and x_r may correspond to x_5 and x_6, respectively. The result z represents an encoding for the stream of two terms corresponding to x, e.g., "y x" in Fig. 2. Then z is decoded by multiplying it by a matrix $\delta = [\delta_\ell; \delta_r] \in \mathbb{R}^{2n \times n}$ and adding a bias vector $\beta_y \in \mathbb{R}^{2n}$, i.e.,

$$y = \delta z + \beta_y \tag{2}$$

The output $y = [\hat{x}_\ell; \hat{x}_r] \in \mathbb{R}^{2n}$ is referred to as the model's *reconstruction* of the input. This model $\theta = \{\varepsilon, \delta, \beta_z, \beta_y\}$ is called an *autoencoder*, and training the model involves measuring the error between the original input vector x and the reconstruction y, i.e.,

$$E(x; \theta) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \tag{3}$$

Concretely, the model is trained by minimizing Eq. (3). Training the model to encode streams with more than two terms requires

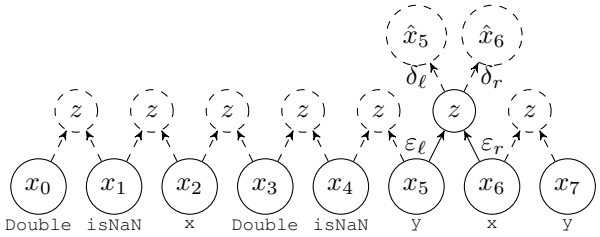

Fig. 2: First iteration of encoding a stream of lexemes

recursively applying the autoencoder. To this end, we use the greedy procedure defined by Socher et al. [71]. In the first iteration of the procedure, each pair of adjacent terms are encoded (Fig. 2). The pair whose encoding yields the lowest reconstruction error (Eq. (3)) is the pair selected for encoding at the current iteration. For example, in Fig. 2, the pair "y x" is selected to be encoded first. As a result, in the next iteration, x_5 and x_6 are replaced by z and the procedure repeats. Upon deriving an encoding for the entire stream, the backpropagation through structure algorithm [72] computes partial derivatives of the (global) error function w.r.t. θ. Then the error signal is optimized using standard methods.

Our intent behind using this learning-based approach is manyfold. First, given that we aimed to assess the effectiveness of using similarities in sorting and transforming fix space elements for patch generation, this recursive learning procedure gave us the ability to evaluate similarity-based sorts at well-defined levels of granularity. The same model can be used to recognize similarities among classes, methods, or even identifiers. Second, the approach did not require intuition in the matter of engineering features for fragments since the approach automatically searched for empirically-based features. Third (and perhaps most important), the learning procedure has been shown to go beyond syntactic similarity [15]. Aside from syntax implicitly organizing the lexemes, the procedure is not syntax directed. In fact, the greedy procedure is free to encode streams of lexemes that are *not* valid programming constructs. Moreover, by initializing the procedure with trained embeddings, it is capable of recognizing similarities among fragments that are only weakly syntactically similar or even syntactically dissimilar fragments with similar functionality (i.e., Type IV clones) [15], which has important implications on prioritizing both textually *and functionally* similar fragments when sorting ingredients.

The third and last stage of our machine learning phase involves clustering identifiers' embeddings. For example, we clustered Math-63 identifiers' embeddings and used t-SNE [73] to reduce their dimensionality $\mathbb{R}^n \rightarrow \mathbb{R}^2$. We plotted a handful of identifiers in our online appendix [26] where terms with similar semantics appear to be proximate in the feature space. In this context, by similar "semantics," we mean terms are used in similar ways in the program (i.e., their token neighborhoods are similar).

Our machine learning phase induces models from source code. These models are then used to make decisions. During the program repair phase, decisions are made on whether or not to transform ingredients and *we use the clusters to operationalize*

the decision criterion. When transforming ingredients, the only identifiers that may replace an out-of-scope variable access are identifiers in the same cluster (Sec. III-C).

C. Program Repair Phase

Core repair loop: DeepRepair is based on a typical generate-and-validate repair loop à la GenProg. First, it begins in a traditional way by running fault localization to get a list of suspicious statements and their suspicious values. These suspicious statements serve as modification points, i.e., points where candidate patches would be applied. For each modification point in sequence, a repair operator is used to transform that statement. Then, one tries to recompile the changed class (since repair operators do not guarantee a well-formed program after modification), and the tentative patch is validated against the whole test suite.

Repair operators: In DeepRepair, the repair operators are "addition of statement" and "replacement of statement." Contrary to GenProg and jGenProg, we do not use statement removal because it generates too many incorrect patches [74]. Addition and replacement are redundancy-based repair operators, since they need to select code from elsewhere in the codebase. In DeepRepair, the reused code are in "ingredient pools," with three main pools, corresponding to whether the ingredient is in the same class as the modification point (local reuse), in the same package (package reuse), or anywhere in the codebase (global reuse). DeepRepair departs from jGenProg on two fundamental points. First, while the default jGenProg operators randomly pick one statement from the ingredient pool, DeepRepair sorts the ingredients according to a specific criterion based on code similarity. Second, the default jGenprog operators reuse code "as is" at the modification point, without applying any transformation, so it could happen that the ingredient has variables that are not in scope at the modification point (or wrongly typed), resulting in an *uncompilable*—obviously incorrect—candidate patch. *On the contrary, DeepRepair has the ability to transform an ingredient so that it fits within the programming scope of the modification point.* In DeepRepair, the sorting and transformation of ingredients is based on deep learning. A combination of a sorting and transformation technique is called a *fix space navigation strategy*. In this paper, we explore five novel navigation strategies implemented in DeepRepair.

Sorting ingredients: When DeepRepair applies a repair operator, it sorts the available repair ingredients. DeepRepair prioritizes the ingredients that come from methods (resp. classes) that are similar to the method (resp. class) containing the modification point. For example, if the suspicious statement is the return statement in Fig. 1, then DeepRepair takes the parent method MathUtils::equals(double,double) and uses the method-level similarity list to sort methods in the codebase. Then, it extracts the statements from each similar method in order and enqueues them in a first-in first-out ingredient queue.

Transforming ingredients: For a patch to be compiled, the ingredient must "fit" in the modification point in the sense that all variable accesses must be in scope. We refer to these "fit" ingredients as *compilable* ingredients. The key point of DeepRepair is its ability to transform ingredients, so that a repair ingredient can be adapted to a particular context, resulting in uncompilable ingredients becoming compilable at the modification point. If a variable is out of scope, then DeepRepair examines the other identifiers *in its cluster* to determine whether any of them are in scope. For example, suppose a patch attempt consists of replacing the return statement in Fig. 1 and suppose we poll the following ingredient: `return equals(x,y,1) || FastMath.abs(y-x) <= eps;` The variable eps is out of scope at the modification point, but SAFE_MIN—a term in its cluster—is in scope, so we replace eps with SAFE_MIN in the ingredient. Applying this patch with a transformed ingredient yields a correct patch for Math-63 (Sec. V-D). Transforming repair ingredients using clusters based on learned embeddings enables DeepRepair to create novel patches. These patches would be impossible to generate if only raw ingredients were used as done in jGenProg. To emphasize the fact that the space of possible DeepRepair patches is *not* the same as jGenProg, DeepRepair's fix space comprises *every* statement in the codebase, which is jGenProg's fix space. However, the ability to transform ingredients means DeepRepair is able to generate patches that are not attainable by jGenProg. DeepRepair's correct patch for Math-63 included a statement that did not exist in Math-63 (Revision ID: d2a5bc0); therefore, jGenProg is not capable of generating this correct patch.

IV. Empirical Validation

In Sec. IV-A, we define our research questions, define the goal of our empirical study, establish the experimental baseline configuration, and state our hypotheses. We conclude our plan with our experimental design: comparative experiments are characterized by treatments, experimental units (i.e., the objects to which we apply the treatments), and the responses that are measured. Then we describe our data collection procedures (Sec. IV-B) and conclude this section by specifying our analysis procedures for each one of our research questions (Sec. IV-C).

A. Experiment Scope and Plan

Our research questions included the following:

- *RQ1* Do code similarities based on deep learning improve fix space navigation as compared to a uniform random search strategy?
- *RQ2* Does ingredient transformation using embeddings based on deep learning effectively transform repair ingredients as compared to a default ingredient application algorithm that does not transform ingredients?
- *RQ3* Does DeepRepair, our learning-based approach that uses code similarities *and* transforms ingredients, improve fix space navigation as compared to jGenProg?
- *RQ4* Does DeepRepair generate higher quality patches than jGenProg?

The goal of our empirical study was to analyze ingredient search strategies for the purpose of evaluation with respect to effectiveness [75]. Our study was from the viewpoint of

TABLE I: DeepRepair configurations evaluated in the study

Code	Features
ED	(E)xecutable-level similarity ingredient sorting (D)efault ingredient application (no ingredient transformation)
TD	(T)ype-level similarity ingredient sorting (D)efault ingredient application (no ingredient transformation)
RE	(R)andom ingredient sorting (E)mbeddings-based ingredient transformation
EE	(E)xecutable-level similarity ingredient sorting (E)mbeddings-based ingredient transformation
TE	(T)ype-level similarity ingredient sorting (E)mbeddings-based ingredient transformation

the software maintainer in the context of six open-source Java projects, a collection of (real) reproducible bugs, and a collection of JUnit test cases [75].

The baseline configuration, jGenProg, was the uniform random search strategy where ingredients were selected from a pool of equiprobable statements. We configured the baseline with a cache so the same modification instance (i.e., ingredient and operator instance) was never attempted more than once to improve its efficiency. The baseline was also configured with a default ingredient application algorithm that analyzed the variable accesses in a repair ingredient, matching accesses' names and types to variables in scope. If at least one variable access failed to match a variable in scope, then the ingredient was discarded. Tab. I lists the DeepRepair configurations. Each configuration evaluates sorting at a particular level of granularity (executable- or type-level) with or without the ability to transform ingredients.

We formalized our experiment into the following hypotheses:

- H_0^{1a} Using code similarities generates the same number of test-adequate patches (on average) as jGenProg.
- H_0^{1b} Using code similarities attempts[1] the same number of ingredients (on average) before finding a test-adequate patch as jGenProg.
- H_0^{2a} Ingredient transformation using embeddings generates the same number of test-adequate patches (on average) as jGenProg.
- H_0^{2b} Ingredient transformation using embeddings attempts the same number of ingredients (on average) before finding a test-adequate patch as jGenProg.
- H_0^{3a} DeepRepair generates the same number of test-adequate patches (on average) as jGenProg.
- H_0^{3b} DeepRepair attempts the same number of ingredients (on average) before finding a test-adequate patch as jGenProg.
- H_0^{4} There is no significant difference in quality between patches generated by DeepRepair and jGenProg.

We chose the following dependent variables: number of test-adequate patches and number of ingredients attempted. We chose the following independent variables (emphasizing the factors): *ingredient search strategy*, *scope*, *clone granularity*, *variable resolution algorithm*, fault localization threshold, maximum number of suspicious candidates, and programming language. The fault localization threshold was fixed at 0.1; the maximum number of suspicious candidates was fixed at 1,000; and the language was fixed on Java. Fixing the language on Java precluded comparing our approach to other popular repair approaches such as Prophet [40], Angelix [34], and SearchRepair [37] that target C programs. The random seed for each experimental configuration ranged from one to three.

We designed comparative experiments to measure the statistical significance of our results. Typically, empirical studies in the field report the number of test-adequate patches found, but we conducted a large-scale study, report the number of test-adequate patches found, and measure the statistical significance of differences to contextualize the results. For our quantitative study, the treatments in our experimental design were the ingredient search strategies. The experimental units were the buggy program revisions, and the responses were the number of test-adequate patches found and the number of ingredients attempted before finding a test-adequate patch.

B. Data Collection Procedure

The first stage of our recognition phase involved creating a model of source code (Sec. III-A). We used Spoon [76], an open-source library for analyzing and transforming Java source code, to build a model for each buggy program revision.

Given the models, we implemented program processors for querying program elements at three levels of granularity: file-, type-, and executable-level granularity. Types included classes and interfaces, and executables included methods and constructors, but we omitted anonymous blocks. We only queried top-level types and executables to control the number of similarities to be computed. In this context, by "top-level," we mean types or executables that did not have a parent type or executable, respectively. For example, a nested class would not be included, but its enclosing class may be included. Then we extracted the yield [47] from each program element's syntax tree along with a key for uniquely identifying the element to build three corpora (Sec. III-A). So each line of a type-level corpus corresponded to a class or interface in the program. The only normalization we performed was replacing characters, floats, integers, and strings with their respective type.

The first stage of our learning phase involved inducing neural network language models from the normalized file-level corpora (Sec. III-B). We used word2vec [77]–[79] to learn embeddings for *each* buggy program revision. We selected word2vec over other architectures because the models can be trained quickly, and we only used the language models' embeddings to initialize the embeddings for the recursive autoencoders rather than randomly initialize them. We used the skip-gram model and set the size of the word vectors to 400 in accordance with previous studies using similar subject systems [15]. We set the maximum skip length between words to 10, used a hierarchical softmax to optimize the computation of output vectors' updates [80], and trained each model for 20 iterations. The language models enabled us to transform the file-level corpus for each program revision into streams of embeddings.

[1] An **attempt** is defined to be a request sent to the fix space for an ingredient.

Then we trained recursive autoencoders to encode streams of embeddings. The encoders used hyperbolic tangent activations (i.e., $f ::= \tanh$ in Eq. (1)), used L-BFGS [81] to optimize costs in batch mode, and trained for up to 50 epochs. After an encoder was trained on a revision's file-level corpus, we used it to encode every type and executable in the revision's type- and executable-level corpora, respectively. Given the encodings, we computed the pairwise Euclidean distance between each pair of types in the type-level corpus and each pair of executables in the executable-level corpus to measure similarities for each program revision.

Next, we extracted the term embeddings from the trained encoder and clustered them using k-means. For each revision, to determine k, we used simulated annealing, initializing both k and the temperature to be the square root of the corresponding vocabulary size. The reason we chose the square root of the vocabulary size is because of its effectiveness in related contexts that categorize words [65]. The objective we optimized was minimizing the number of points with negative silhouette values. Thus, at the end of the learning phase, each buggy revision had a cached list of executable- and type-level similarities as well as a categorization for identifiers.

The final phase of our technical approach involved automatically repairing buggy program revisions (Sec. III-C). Our subject systems comprised six open-source Java projects including 374 buggy program revisions in Defects4J database version 1.1.0 [23], [24].[2] In Tab. II, we report median values since each project has several buggy program revisions.

To run repair experiments, we used Astor [21], an automatic software repair framework for Java. Within the Astor framework, we leveraged GZoltar [82], a spectrum-based fault localization tool, to compute the Ochiai formula [83] for statements' suspicious values.

Each trial corresponded to a seeded treatment, which was a factorial of strategy and scope. Empirical studies indicated that fragment locality matters in software maintenance and evolution, so we analyzed three different levels of scope: local, package, and global [6]. For local scope, Astor builds the ingredient search space by amalgamating the distinct set of classes that contain at least one suspicious statement. For package scope, Astor computes the distinct set of packages that contain at least one suspicious statement and builds the ingredient search space using the set of classes in those packages. For global scope, Astor builds the ingredient space using all classes from the application under repair. We configured Astor to *not* stop at the first patch found and—starting from the original program—to continue searching for other patches until reaching a three-hour time limit. In total, we ran 20,196 trials (374 buggy program revisions $\times$ 6 search strategies $\times$ 3 levels of scope $\times$ 3 random seeds) on subclusters comprising 64 compute nodes running Red Hat Enterprise Linux 6.2. Each compute node was a Dell PowerEdge C6100 serving two Intel Xeon X5672 quad-core processors at 3.2 GHz with 12 MB L3 cache and at least 48 GB of 1333 MHz main memory. Each trial was allocated two (hyper-threaded) cores and evolved one program variant for three hours using three repair operators: InsertAfterOp, InsertBeforeOp, and ReplaceOp. *We did not include RemoveOp in our operator space because we only focused on repair operators that reuse code*. We also wanted to guard against meaningless patches that simply remove functionality.

[2]We could not build the Spoon model for Mockito bugs 1–21, which was likely because of missing or incompatible dependencies.

TABLE II: Project statistics

Project	Files	LOC	Tokens	Vocab.
Apache commons-lang	221	48,890	420,000	4,672
Apache commons-math	845	97,130	830,000	8,450
Closure compiler	937	247,300	1,449,000	26,490
JFreechart	952	130,300	921,800	9,008
Joda-Time	316	81,640	736,300	5,989
Mockito	680	44,990	309,500	5,735

C. Analysis Procedure

RQ1: We analyzed the effectiveness of fix space navigation strategies in two parts. The first part analyzed effectiveness at generating test-adequate patches. We used the non-parametric Wilcoxon test with a Bonferroni correction to compare the number of test-adequate patches using jGenProg versus the strategy using code similarities at executable- and type-level granularity (i.e., ED and TD in Tab. I). We used Wilcoxon since the test-adequate patch counts could be paired, and we used a Bonferroni correction since we performed several tests simultaneously at different levels of scope and strategy. To complement our statistical analysis on number of test-adequate patches found, we also computed the difference between the set of jGenProg patches and the set of DeepRepair patches. Specifically, if D is the set of DeepRepair patches, and J is the set of jGenProg patches, then we computed $|D \setminus J| \,/\, |D|$, the percentage of DeepRepair patches that were not found by jGenProg. The second part analyzed the number of attempts to generate test-adequate patches. We used the non-parametric Mann-Whitney test with a Bonferroni correction to compare the number of attempts to generate test-adequate patches using jGenProg and using code similarities at executable- and type-level granularity. To complement our analysis, we also plotted the number of attempts to generate compilable ingredients.

RQ2: We analyzed effectiveness in two parts. We used the Wilcoxon test to compare the number of test-adequate patches using jGenProg versus the uniform random search strategy with the embeddings-based ingredient transformation algorithm (i.e., RE in Tab. I). The algorithm gives jGenProg the ability to transform repair ingredients containing variable accesses that are out of scope. We also computed the difference between the set of jGenProg patches and the set of DeepRepair patches. Additionally, we used the Mann-Whitney test to compare the number of attempts to generate test-adequate patches and plotted the number of attempts to generate compilable ingredients.

RQ3: Our experimental design for *RQ3* was virtually identical to our design for *RQ1* except here we compared jGenProg to the strategy using code similarities with the

embeddings-based ingredient transformation algorithm (i.e., EE and TE in Tab. I).

RQ4: We used correctness as a proxy for quality. Three judges evaluated the same random sample of 30 (15 jGenProg and 15 DeepRepair) patches to assess correctness. Martinez et al. [84] defined the *correctness* of a patch to be one of three values: correct, incorrect, or unknown. *Correct* denotes a patch is equivalent (according to the judge's understanding) to the human-written patch. Judges were also prompted for their confidence in their correctness rating where confidence was one of four values: high, moderate, slight, and none. Additionally, for reproducibility, judges also assessed the readability of each patch, where the *readability* of a patch was either easy, medium, or hard in accordance with previous studies on patch correctness [25], [85]. We define readability to be the subjective qualification of how easily the patch can be understood. Readability is a subjective aggregation of different quantitative metrics: patch size in number of lines, number of involved variables, number of method calls, and the types of AST elements being inserted or replaced. There is no accepted quantitative aggregation of these metrics, and we think that a quantitative aggregation would be project- and even bug-dependent. In addition to the random sample, judges also evaluated the patches generated by jGenProg that were not found by DeepRepair and patches generated by DeepRepair that were not found by jGenProg.

V. Empirical Results

We ran 20,196 repair trials, 247 of which were killed. The 19,949 trials that finished took a total of 2,616 days of computation time. Zero patches were found for Closure, Mockito, and Time bugs. The baseline configuration found test-adequate patches for 48 different bugs. Six other bugs were unlocked by DeepRepair configurations. The trials found 19,832 different test-adequate patch *instances* and attempted 406,443,249 ingredients.

A. RQ1 (Analysis of ED and TD Strategies)

Tab. III lists the bugs for which test-adequate patches were found at (L)ocal, (P)ackage, and (G)lobal scope. jGenProg found test-adequate patches for 48 bugs. The treatments ED and TD found patches for 40 and 38 bugs, respectively.

Since many configurations found more than one patch for the same bug, we compared the patch counts between jGenProg and the two treatments ED and TD to see whether one configuration was more productive than another. We failed to reject the null hypothesis H_0^{1a} at each level of scope. Next, we analyzed the sets of patches and observed approximately 99%, 25%, and 36% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. This result means that DeepRepair finds alternative patches.

We also counted the number of attempts to find each patch. Fig. 3 shows descriptive statistics for the number of attempts to find a test-adequate patch. We failed to reject the null hypothesis H_0^{1b} at each level of scope. We also counted the number of attempts to find each compilable ingredient (as defined in Sec. III-C). Fig. 4 shows descriptive statistics for the number of attempts to find a compilable ingredient at each level of scope for jGenProg, ED, and TD. Generally, sorting the fix space using code similarities results in fewer attempts before finding a compilable ingredient.

Key result. DeepRepair's search strategy using code similarities generally finds compilable ingredients faster than the baseline, but this improvement neither yields test-adequate patches in fewer attempts (on average) nor finds significantly

TABLE III: Patches found at (L)ocal, (P)ackage, and (G)lobal scope

ProjectID	BugID	jGenProg	ED	TD	RE	EE	TE
Chart	1	LPG	LPG	LPG	LPG	LPG	LPG
	3	LPG	LPG	LPG	LPG	LPG	LPG
	5	LPG	LPG	LPG	LPG	LPG	LPG
	7	LPG	LPG	LPG	LPG	LPG	LPG
	9	-	-	-	-	P	-
	12	G	LPG	LPG	G	LPG	LPG
	13	LPG	LPG	LPG	LPG	LPG	LPG
	14	LPG	LPG	LPG	LPG	LPG	LPG
	15	LPG	LPG	LPG	LPG	LPG	LPG
	18	L	-	-	L	-	-
	25	LPG	LPG	LPG	LPG	LPG	-
	26	LPG	LPG	LPG	LPG	LPG	LPG
Lang	7	LPG	LPG	LPG	LPG	LPG	LPG
	10	LPG	LPG	LPG	LPG	LPG	LPG
	20	LPG	LPG	LPG	LPG	LPG	LPG
	22	LPG	LPG	LPG	LPG	LPG	LP
	24	LPG	LPG	L	LPG	L	L
	27	LPG	LPG	LPG	LPG	LPG	LPG
	38	PG	-	-	PG	-	-
	39	LPG	LPG	LPG	LPG	L	-
Math	2	LPG	LPG	LPG	LPG	LPG	LPG
	5	LPG	LPG	LPG	LPG	LPG	LPG
	6	LP	LPG	LPG	LP	LP	G
	7	L	-	-	-	-	-
	8	-	-	-	G	LPG	LPG
	18	L	-	-	P	-	-
	20	LPG	LPG	LPG	LPG	LPG	LPG
	22	LPG	LPG	-	LPG	LPG	-
	24	-	-	-	LP	-	-
	28	LPG	LPG	LPG	LPG	LPG	LPG
	32	L	LPG	-	LP	-	-
	40	LPG	LPG	LP	LPG	LPG	L
	44	P	-	-	-	-	-
	49	LPG	LPG	LPG	LPG	LPG	LPG
	50	LPG	LPG	LP	LPG	LP	LP
	53	LPG	LPG	LPG	LPG	LPG	LPG
	56	L	LPG	LP	LPG	LP	-
	57	G	L	LP	G	-	LP
	58	-	-	-	LP	-	-
	60	G	LP	LPG	G	LP	LP
	63	-	-	-	LPG	LPG	LPG
	64	L	-	-	-	-	-
	70	LPG	LPG	LPG	LPG	LPG	LPG
	71	LPG	-	-	LPG	-	-
	73	LPG	LPG	LPG	LPG	LPG	LPG
	74	PG	-	-	PG	-	-
	77	LPG	L	LPG	LPG	L	LPG
	78	LPG	LPG	LPG	LPG	LPG	LPG
	80	LPG	LPG	LPG	LPG	P	LPG
	81	LPG	LPG	LPG	LPG	LPG	LPG
	82	-	-	-	-	PG	G
	84	LPG	LP	LP	LPG	LP	LP
	85	LPG	LPG	LPG	LPG	LPG	LPG
	98	LPG	LPG	LPG	LPG	LPG	LPG
Total	54	48	40	38	49	42	38

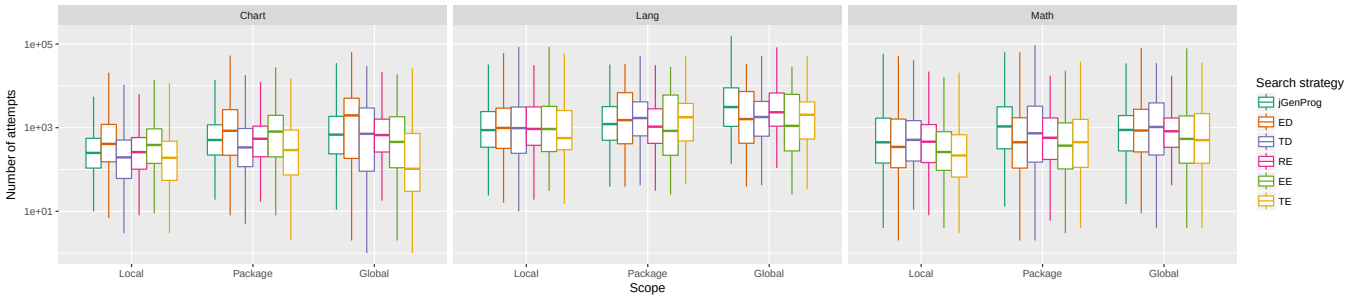

Fig. 3: Number of attempts to find a test-adequate patch

more patches (on average) than the baseline. Yet there were notable differences between DeepRepair and jGenProg patches.

B. RQ2 (Analysis of RE Strategy)

The treatment RE found test-adequate patches for 49 bugs. Comparing the patch counts, we failed to reject the null hypothesis H_0^{2a} at each level of scope. Analyzing the set difference, 53%, 3%, and 53% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. We failed to reject the null hypothesis H_0^{2b} at each level of scope. This can also be graphically seen in Fig. 3 that shows descriptive statistics for the number of attempts to find a test-adequate patch for jGenProg and RE.

Key result. DeepRepair's search strategy using the embeddings-based ingredient transformation algorithm neither yields patches in fewer attempts (on average) nor finds significantly more patches (on average) than jGenProg, but there were notable differences between DeepRepair and jGenProg patches.

C. RQ3 (Analysis of EE and TE Strategies)

The treatments EE and TE found test-adequate patches for 42 and 38 bugs, respectively. Comparing the patch counts, we failed to reject the null hypothesis H_0^{3a} at each level of scope. Analyzing the set difference, 99%, 28%, and 51% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. Fig. 3 shows descriptive statistics for the number of attempts to find a test-adequate patch for jGenProg, EE, and TE. We failed to reject the null hypothesis H_0^{3b} at each level of scope. Fig. 4 shows descriptive statistics for the number of attempts to find a compilable ingredient at each level of scope for jGenProg, EE, and TE.

Key result. DeepRepair's search strategy using code similarities with the embeddings-based ingredient transformation algorithm generally finds compilable ingredients faster than jGenProg, but this improvement neither yields test-adequate patches in fewer attempts (on average) nor finds significantly more patches (on average) than jGenProg. Once more, DeepRepair appears to find a complementary set of patches.

D. RQ4 (Manual Assessment)

After independently evaluating each sample patch, three judges discussed and resolved conflicts in terms of correctness. Five DeepRepair patches and five jGenProg patches were evaluated to be correct. We failed to reject the null hypothesis H_0^4. Although judges did notice differences in the patches generated by the approaches, no significant difference in readability was reported.

In addition to the random sample, we manually examined the following specific sets of patches. There were three bugs patched exclusively by jGenProg. We examined jGenProg's patches for these bugs and found that all of them were clearly incorrect. On the other hand, there were six bugs patched exclusively by DeepRepair configurations. We also examined these patches and report some of our findings below.

Chart-9. The human-written patch was

```
+ if (endIndex < 0 || endIndex < startIndex) {
- if (endIndex < 0) {
```

None of the identifiers in the human-written patch were new (cf. Sec. II-B), but the conditional expression

```
(endIndex < 0 || endIndex < startIndex)
```

was novel w.r.t. the codebase, so generate-and-validate techniques that cannot generate new code would never find this patch. The selection statement that DeepRepair generated passed the test suite, but it was incorrect (to the best of our knowledge). Notably, the DeepRepair patch contained the conditional expression

```
(endIndex < 1 || endIndex > LAST_WEEK_IN_YEAR)
```

whose syntactic structure resembled the human-written expression. DeepRepair's conditional expression was novel w.r.t. the codebase as it was generated by transforming an ingredient. The expression was originally

```
(result < 1 || result > LAST_WEEK_IN_YEAR)
```

but DeepRepair recognized *similarities* in how the identifiers, result and endIndex, were used in the codebase, so it replaced result—a variable out of scope—with endIndex.

Math-63. The human-written patch was

```
  public static boolean equals(double x, double y)
-     return Double.isNaN(x) && Double.isNaN(y) || x == y;
+     return equals(x,y,1);
```

Again, none of the identifiers in the human-written patch were new, but the statement was novel w.r.t. the codebase. DeepRepair generated the following patch:

```
  public static boolean equals(double x, double y)
-     return Double.isNaN(x) && Double.isNaN(y) || x == y;
+     return equals(x,y,1) || FastMath.abs(y-x) <= SAFE_MIN;
```

The ingredient was selected from a *similar* method:

```
public static boolean equals(double x, double y, double eps)
    return equals(x,y,1) || FastMath.abs(y-x) <= eps;
```

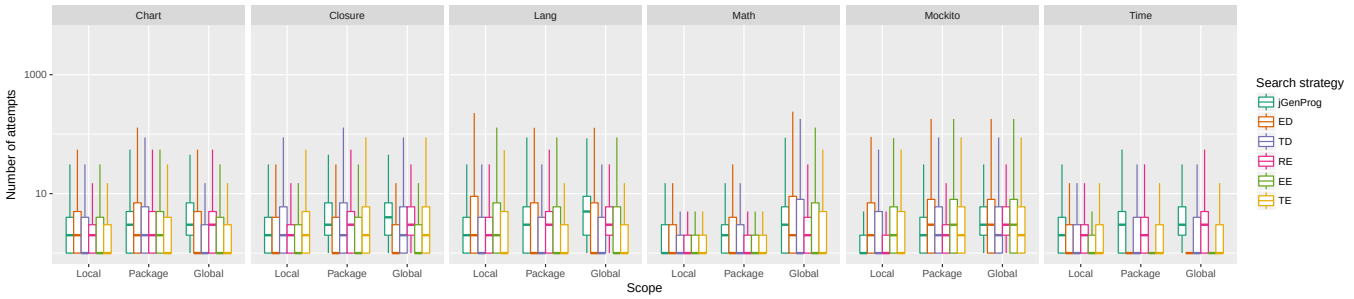

Fig. 4: Number of attempts to find a compilable ingredient

However, the variable eps in the ingredient was not in scope at the modification point. DeepRepair recognized similarities between how the identifiers eps and SAFE_MIN are used in the codebase, so it replaced eps with SAFE_MIN. As a result, both the human-written patch and DeepRepair's patch invoke equals(x,y,1) which returns true if x and y are equal or within the range of allowed errors (inclusive). In this case, the range of allowed error is defined to be zero floating point numbers between the two values, so the values must be the same floating point number or adjacent floating point numbers. Therefore, when equals(x,y,1) is true, both the human-written patch and the DeepRepair patch return true since the conditional expression in the DeepRepair patch short-circuits. When equals(x,y,1) is false, the human-written patch returns false, and since SAFE_MIN is defined to be the smallest normalized number in IEEE 754 arithmetic (i.e., 0x1.0p-1022), the DeepRepair patch returns false. Hence, the DeepRepair patch is semantically equivalent to the human patch and considered correct.

Key result. There are apparent, critical differences observed in DeepRepair's patches compared to jGenProg, which unlock new bugs—that would otherwise have not been patched—by reusing *and transforming* repair ingredients. Our future work aims to extensively analyze more results to understand which defect *classes* can be unlocked with DeepRepair's search strategies.

VI. Threats to Validity

Internal validity: DeepRepair relies on deep learning to compute similarities among code elements at different levels of granularity. Learning-based code clone detection has been evaluated at multiple levels of granularity with promising results [15]. Additionally, we manually examined small random samples of similar code fragments at executable- and type-level granularity from each project to validate some degree of textual/functional similarity in accordance with previous studies. However, we do not claim to have used optimal settings for training on each program revision or even each project. We also acknowledge the confounding configuration choice problem [86]. We did not adopt arbitrary configurations and tried to justify each configuration in our approach.

External validity: In our experiments we evaluate DeepRepair on 374 buggy program revisions (in six unique software systems) from the Defects4J benchmark. One threat is that the number of bugs may not be large enough to represent the actual differences between DeepRepair and jGenProg.

Construct validity: Our empirical evaluation is similar to all previous studies on program repair in that DeepRepair and jGenProg do not target buggy program revisions with multiple faults. Also, prior work noted some impact of flaky test cases on program repair [25]. While we did not detect any flaky tests in our benchmarks during the experiments, their potential presence could impact both DeepRepair and jGenProg.

Conclusion validity: DeepRepair and jGenProg have random components, so different runs would possibly produce different patches. However, our experiments were as computationally extensive as they could have been within our means, consisting of 19,949 trials spanning 2,616 days of computation time. We could not manually analyze all the generated patches as this would require years of manual work, but we randomly sampled a subset for manual evaluation. Three judges inspected each sample to minimize bias.

VII. Conclusion

We introduced a novel learning-based algorithm to intelligently select and transform repair ingredients in a generate-and-validate repair loop based on the redundancy assumption. DeepRepair takes a novel perspective on the ingredient selection problem: it selects ingredients from similar methods or classes where similarity has been inferred with deep unsupervised learning. Many repairs need to replace identifiers to make an ingredient compilable at a specific modification point. To the best of our knowledge, DeepRepair is the first approach that expands the fix space by transforming ingredients using identifiers' similarities. We conducted a computationally intensive empirical study and found that DeepRepair did not significantly improve effectiveness using a new metric (number of attempts), but DeepRepair did generate many patches that cannot be generated by existing redundancy-based repair techniques.

Acknowledgment

We thank David Nader Palacio for his help with conducting the empirical study. This work was performed using computing facilities at the College of William and Mary which were provided by contributions from the National Science Foundation, the Commonwealth of Virginia Equipment Trust Fund, and the Office of Naval Research. This material is based upon work supported by the National Science Foundation under Grant No. 1525902. This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

REFERENCES

[1] D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. ICPC'09.

[2] M. Gabel and Z. Su. A study of the uniqueness of source code. FSE'10.

[3] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. ICSE'12.

[4] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. ICSE'13.

[5] H. Nguyen, A. Nguyen, T. Nguyen, T. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. ASE'13.

[6] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. ICSE Companion'14.

[7] E. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. FSE'14.

[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *TSE*, 38(1), 2012.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE'12.

[10] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. PLDI'15.

[11] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. MSR'15.

[12] C. Corley, K. Damevski, and N. Kraft. Exploring the use of deep learning for feature location. ICSME'15.

[13] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. QRS'15.

[14] A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. ASE'15.

[15] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. ASE'16.

[16] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. ICSE'16.

[17] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. FSE'16.

[18] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Deep learning similarities from different representations of source code. MSR'18.

[19] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. ASE'18.

[20] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. ICSE'19.

[21] M. Martinez and M. Monperrus. Astor: A program repair library for Java (demo). ISSTA'16.

[22] R. Socher. *Recursive Deep Learning for Natural Language Processing and Computer Vision*. PhD thesis, 2014.

[23] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? FSE'14.

[24] R. Just, D. Jalali, and M. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. ISSTA'14.

[25] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *EMSE*, 2016.

[26] Online appendix: https://sites.google.com/view/deeprepair, 2017.

[27] M. Monperrus. Automatic software repair: A bibliography. Technical report, Inria, 2015.

[28] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. GECCO'12.

[29] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *TSE*, 41(12):1236–1256, 2015.

[30] J. Yi, U. Ahmed, A. Karkare, S. Tan, and A. Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. FSE'17.

[31] T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. ICSE'13.

[32] S. Mechtaev, Y. Jooyong, and A. Roychoudhury. DirectFix: Looking for simple program repairs. ICSE'15.

[33] J. Xuan, M. Martínez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, Daniel Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *TSE*, 43(1):34–55, 2016.

[34] S. Mechtaev, Y. Jooyong, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. ICSE'16.

[35] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in C. FSE'17.

[36] X. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. FSE'17.

[37] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. ASE'15.

[38] H. Yokoyama, Y. Higo, K. Hotta, T. Ohta, K. Okano, and S. Kusumoto. Toward improving ability to repair bugs automatically: A patch candidate location mechanism using code similarity. SAC'16, pages 1364–1370, 2016.

[39] M. Martinez and M. Monperrus. ASTOR: Evolutionary automatic software repair for Java. *CoRR*, abs/1410.6651, 2014.

[40] F. Long and M. Rinard. Automatic patch generation by learning correct code. POPL'16.

[41] Y. Bengio, A. Courville, and P. Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[42] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553), 2015.

[43] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

[44] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. Sk_p: A neural program corrector for MOOCs. SPLASH Companion 2016.

[45] R. Gupta, S. Pal, A.Kanade, and S. Shevade. Deepfix: Fixing common C language errors by deep learning. AAAI'17.

[46] http://commons.apache.org/proper/commons-math/.

[47] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. 2 edition, 2006.

[48] D. Jurafsky and J. Martin. *Speech and Language Processing*. 2 edition, 2009.

[49] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. ICST'13.

[50] A. Nguyen, T. Nguyen, and T. Nguyen. Lexical statistical machine translation for language migration. FSE'13.

[51] A. Nguyen, T. Nguyen, and T. Nguyen. Migrating code with statistical machine translation. ICSE Companion'14.

[52] P. Tonella, R. Tiella, and D. Nguyen. Interpolated n-grams for model based testing. ICSE'14.

[53] J. Campbell, A. Hindle, and J. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. MSR'14.

[54] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. FSE'14.

[55] A. Nguyen, H. Nguyen, T. Nguyen, and T. Nguyen. Statistical learning approach for mining API usage mappings for code migration. ASE'14.

[56] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. PLDI'14.

[57] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. CACHECA: A cache language model based code suggestion tool. ICSE'15.

[58] V. Hellendoorn, P. Devanbu, and A. Bacchelli. Will they like this? Evaluating code contributions with language models. MSR'15.

[59] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. ICSE'16.

[60] T. Nguyen, H. Pham, P. Vu, and T. Nguyen. Learning api usages from bytecode: A statistical approach. ICSE'16.

[61] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. ASE'16.

[62] V. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? FSE'17.

[63] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *JMLR*, 3, 2003.

[64] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur. Recurrent neural network based language model. INTERSPEECH'10.

[65] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocký. RNNLM - Recurrent neural network language modeling toolkit. ASRU'11.

[66] T. Mikolov, S. Kombrink, L. Burget, J. Cernocký, and S. Khudanpur. Extensions of recurrent neural network language model. ICASSP'11.

[67] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocký. Strategies for training large scale neural network language models. ASRU'11.

[68] T. Mikolov. *Statistical Language Models Based on Neural Networks*. PhD thesis, 2012.

[69] R. Socher, J. Pennington, E. Huang, A. Ng, and C. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. EMNLP'11.

[70] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. EMNLP'13.

[71] R. Socher, C. Lin, A. Ng, and C. Manning. Parsing natural scenes and natural language with recursive neural networks. ICML'11.

[72] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. ICNN'96.

[73] L. van der Maaten and G. Hinton. Visualizing high-dimensional data using t-SNE. *JMLR*, 9:2579–2605, 2008.

[74] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. ICSE'14.

[75] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. 2000.

[76] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *SPE*, 46:1155–1179, 2015.

[77] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[78] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. NIPS'13.

[79] T. Mikolov, W. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. NAACL-HLT'13.

[80] X. Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[81] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151), 1980.

[82] J. Campos, A. Riboira, A. Perez, and R. Abreu. GZoltar: An eclipse plug-in for testing and debugging. ASE'12.

[83] R. Abreu, P. Zoeteweij, and A. van Gemund. An evaluation of similarity coefficients for software fault localization. PRDC'06.

[84] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *EMSE*, 20(1):176–205, 2015.

[85] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. ISSTA'15.

[86] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. FSE'13.